```
#r "nuget: FSharp.Stats, 0.4.1"
#r "nuget: FSharp.Data"

#load "../common.fsx"

open System
open FSharp.Data
open Common

open FSharp.Stats


Environment.CurrentDirectory <- __SOURCE_DIRECTORY__
```

Portfolio Optimization

We're now going to see how to do mean-variance portfolio optimization. The objective is to find the portfolio with the greatest return per unit of standard deviation.

In particular, we're going to identify the tangency portfolio. The tangency portfolio is the portfolio fully invested in risky assets that has the maximum achievable sharpe ratio. When there is a risk-free rate, the efficient frontier of optimal portfolios is some combination of the tangency portfolio and the risk-free asset. Investors who want safe portfolios hold a lot of bonds and very little of the tangency portfolio. Investors who want riskier portfolios hold little risk-free bonds and a lot of the tangency portfolio (or even lever the tangency portoflio).

Now one thing to keep in mind is that often you think of this as the optimal weight per security. But one well known problem is that trying to do this naively does not work well. And by naively I mean taking a stock's average return and covariances in the sample. In large part, this is because it is hard to estimate a stock's past returns. I know. Big shock, right?

However, there are ways to do portfolio optimization that works better. We can do it by creating large groups of stocks with similar characteristics. For example, a factor portfolio. Then you estimate the expected return and covariance matrix using the factor. That tends to give you better portfolios because the characteristics that you're using to form the portfolios help you estimate the return and covariances of the stocks in it.

A type to hold our data.

```
type StockData =
    { Symbol : string
      Date : DateTime
      Return : float }
```

We get the Fama-French 3-Factor asset pricing model data.

```
let ff3 = French.getFF3 Frequency.Monthly

// Transform to a StockData record type.
let ff3StockData =
    [|
       ff3 |> Array.map(fun x -> {Symbol="HML";Date=x.Date;Return=x.Hml})
       ff3 |> Array.map(fun x -> {Symbol="MktRf";Date=x.Date;Return=x.MktRf})
       ff3 |> Array.map(fun x -> {Symbol="Smb";Date=x.Date;Return=x.Smb})
    |] |> Array.concat
```

Let's get our factor data.

```
let myFactorPorts = CsvProvider<"myExcessReturnPortfolios.csv",
                                ResolutionFolder = __SOURCE_DIRECTORY__>.GetSample()

let long =
    myFactorPorts.Rows
    |> Seq.filter(fun row -> row.PortfolioName = "Mine" && row.Index = Some 3)
    |> Seq.map(fun x -> { Symbol = "Long"; Date = x.YearMonth; Return = x.Ret })
    |> Seq.toArray
```

Some good standard investments.

```
let vti =
    "VTI"
    |> Tiingo.request
    |> Tiingo.startOn (DateTime(2000,1,1))
    |> Tiingo.getReturns

let bnd =
    "BND"
    |> Tiingo.request
    |> Tiingo.startOn (DateTime(2000,1,1))
    |> Tiingo.getReturns
```

These are daily returns. So let's convert them to monthly returns.

Remember how to do this?

```
let exampleRets = [| 0.1; 0.1; 0.1 |]

(1.0, exampleRets)
||> Array.fold (fun acc ret -> acc * (1.0 + ret))
|> fun grossReturn -> grossReturn - 1.0
```

```
```

```
// Compare to
1.1**3.0 - 1.0
```

So let's combine the `VTI` and `BND` data, group by symbol and month, and then convert to monthly returns.

```
let standardInvestments =
    Array.concat [vti; bnd]
    |> Array.groupBy(fun x -> x.Symbol, x.Date.Year, x.Date.Month)
    |> Array.map(fun ((sym, year, month), xs) ->
        let sortedRets =
            xs
            |> Array.sortBy(fun x -> x.Date)
            |> Array.map(fun x -> x.Return)
        let monthlyGrossRet =
            (1.0, sortedRets)
            ||> Array.fold (fun acc x -> acc * (1.0 + x))
        { Symbol = sym
          Date = DateTime(year, month, 1)
          Return = monthlyGrossRet - 1.0 })
```

And let's convert to excess returns

```
let rf = ff3 |> Seq.map(fun x -> x.Date, x.Rf) |> Map

let standardInvestmentsExcess =
    let maxff3Date = ff3 |> Array.map(fun x -> x.Date) |> Array.max
    standardInvestments
    |> Array.filter(fun x -> x.Date <= maxff3Date)
    |> Array.map(fun x ->
        match Map.tryFind x.Date rf with
        | None -> failwith $"why isn't there a rf for {x.Date}"
        | Some rf -> { x with Return = x.Return - rf })
```

If we did it right, the `VTI` return should be pretty similar to the `MktRF` return from Ken French's website.

```
standardInvestments
|> Array.filter(fun x -> x.Symbol = "VTI" && x.Date.Year = 2021)
|> Array.map(fun x -> x.Date.Month, round 4 x.Return)
|> Array.take 3
```

```
val it : (int * float) [] = [|(1, -0.0033); (2, 0.0314); (3, 0.0365)|]
```

```
ff3
|> Array.filter(fun x -> x.Date.Year = 2021)
|> Array.map(fun x -> x.Date.Month, round 4 x.MktRf)
```

```
val it : (int * float) [] = [|(1, -0.0003); (2, 0.0278); (3, 0.0309)|]
```

Let's put our stocks in a map keyed by symbol

```
let stockData =
    Array.concat [| standardInvestmentsExcess; long |]
    |> Array.groupBy(fun x -> x.Symbol)
    |> Map

let symbols =
    stockData
    |> Map.toArray // convert to array of (symbol, observations for that symbol)
    |> Array.map fst // take just the symbol
    |> Array.sort // sort them
```

Let's create a function that calculates covariances for two securities.

```
let getCov x y stockData =
    let innerJoin xId yId =
        let xRet = Map.find xId stockData
        let yRet = Map.find yId stockData |> Array.map(fun x -> x.Date, x) |> Map
        xRet
        |> Array.choose(fun x ->
            match Map.tryFind x.Date yRet with
            | None -> None
            | Some y -> Some (x.Return, y.Return))
    let x, y = innerJoin x y |> Array.unzip
    Seq.cov x y

let covariances =
    symbols
    |> Array.map(fun x ->
        symbols
        |> Array.map(fun y -> getCov x y stockData))
    |> matrix
let means =
    stockData
    |> Map.toArray
    |> Array.map(fun (sym, xs) ->
        sym,
        xs |> Array.averageBy(fun x -> x.Return))
    |> Array.sortBy fst
    |> Array.map snd
    |> vector
```

This solution method for finding the tangency portfolio comes from Hilliar, Grinblatt and Titman 2nd European Edition, Example 5.3.

Since it has the greatest possible Sharpe ratio, that means that you cannot rebalance the portfolio and increase the return per unit of standard deviation.

The solution method relies on the fact that covariance is like marginal variance. At the tangency portfolio, it must be the case that the ratio of each asset's risk premium to it's covariance with the tangency portfolio, $(r_i - r_f)/cov(r_i, r_p)$, must be the same. Because that's the return per marginal variance ratio, and if it was not equal for all assets, then you could rebalance and increase the portfolio's return while holding the portfolio variance constant.

In the below algebra, we solve for the portfolio that has covariances with each asset equal to the asset's risk premium. Then we relever to have a portfolio weight equal to 1.0 (we can relever like this because everything is in excess returns) and then we are left with the tangency portfolio.

```
// solve A * x = b for x
let w' = Algebra.LinearAlgebra.SolveLinearSystem covariances means
let w = w' |> Vector.map(fun x -> x /  Vector.sum w')
```

Portfolio variance

```
let portVariance = w.Transpose * covariances * w
```

Portfolio standard deviation

```
let portStDev = sqrt(portVariance)
```

Portfolio mean

```
let portMean = w.Transpose * means
```

Annualized Sharpe ratio

```
sqrt(12.0)*(portMean/portStDev)
```